Distributed coordination service: Zookeeper

Suresh Podeti

4 min read · Nov 5, 2023

Introduction

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for:

- **Synchronisation**
- **Configuration maintenance**
- **Groups and naming**

It uses a data model similar to directory **tree structure of file systems**.

It runs in Java and has bindings for both Java and C.

Hard part

Coordination services are hard to get it right. They are especially prone to errors such as:

- **Race conditions**
- **Deadlock**

The motivation behind ZooKeeper is to **relieve** distributed applications the responsibility of implementing coordination services **from scratch**.

Design Goals

Zookeeper is simple

ZooKeeper allows distributed processes to coordinate with each other through a **shared hierarchical namespace** which is organised similarly to a standard file system. The namespace consists of data registers- called *znodes*.

A name is a sequence of path elements separated by a slash (/). Every node in ZooKeeper's namespace is identified by a path. Unlike standard file systems, each node in a ZooKeeper namespace can have data associated with it as well as children. **It is like having a file-system that allows a file to also be a directory.** Zookeeper was designed to store coordination data like:

- Status information
- Configuration
- Location information

Data stored at each node is usually small, in the byte to kilobyte range. Zookeeper data is kept **in-memory**, which means Zookeeper can achieve high throughput and low latency numbers.

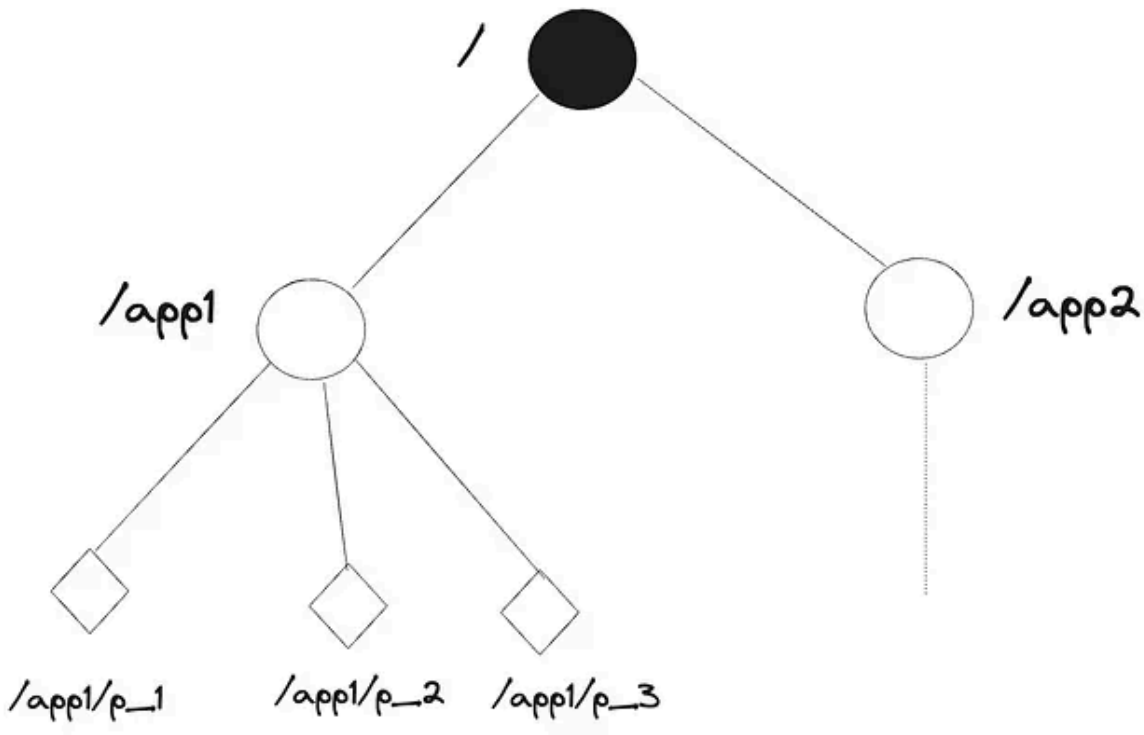

Fig 1.0: Data model and the hierarchical namespace for zookeeper

Zookeeper is replicated

Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a set of hosts called **ensemble**.

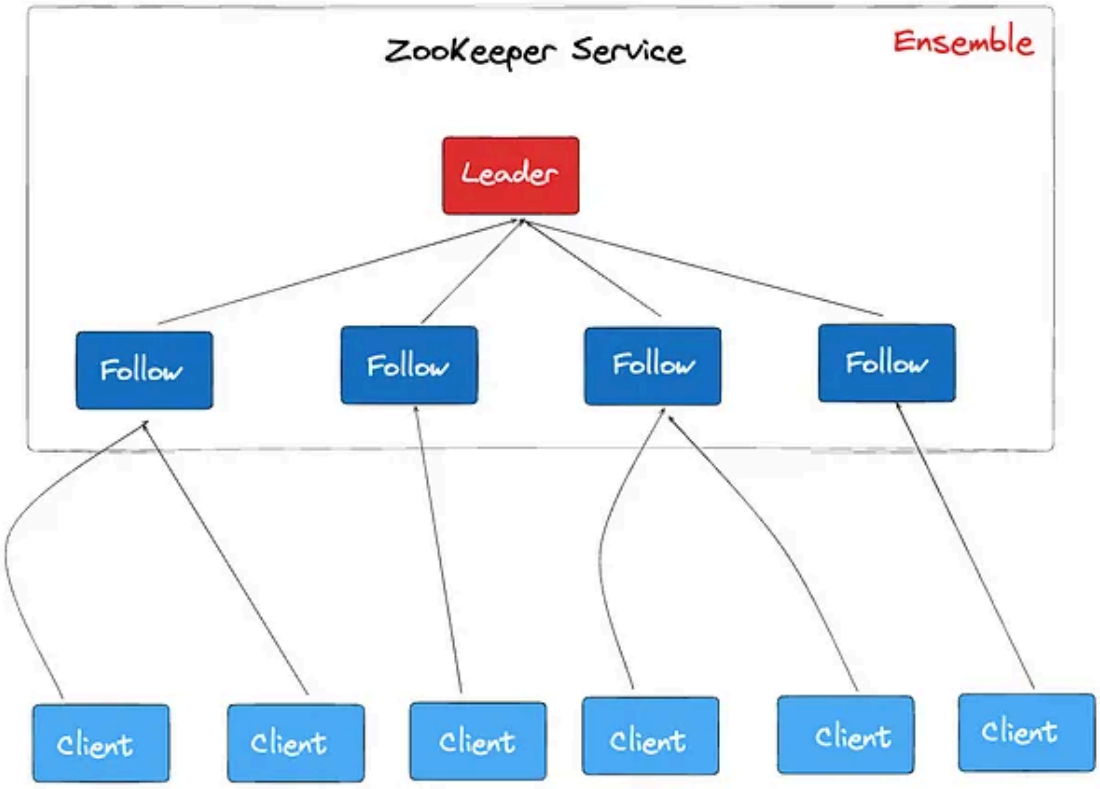

Fig 2.0: Zookeeper ensemble

The server that make up the ZooKeeper service must all know about each other, and it is not sufficient for only the leader to know about the all the servers in the ensemble because this knowledge is fundamental to the system's operation.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Zookeeper is ordered

ZooKeeper stamps each update with a number that reflects the order of all ZooKeeper transactions.

Zookeeper is fast

It is especially fast in *read-dominant* workloads. Zookeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.

Znodes

- **Persistent Nodes**: These are regular znodes that remain in the data tree until they are explicitly deleted by a client. They are often used for storing long-term configuration information or representing the state of various aspects of a distributed system.

- **Ephemeral Nodes**: Ephemeral znodes are associated with a client's session and exist only as long as that session is active. **When** the client's **session ends** (e.g., due to a disconnection or expiration), the **ephemeral nodes** created by that client are **automatically deleted**. Ephemeral nodes are useful for representing the temporary presence of a client or a resource.

- **Sequential Nodes**: Sequential znodes are similar to persistent nodes but have a unique, **monotonically increasing identifier** as part of their name. These identifiers are generated by ZooKeeper and help in ordering znodes based on the order of their creation. Sequential nodes are often used for creating ordered lists or implementing distributed queues.

- **Sequential Ephemeral Nodes**: These znodes combine the properties of both sequential and ephemeral nodes. They have a unique identifier, like sequential nodes, and are also automatically deleted when the client's session ends, similar to ephemeral nodes. Sequential ephemeral nodes are frequently used for tasks like distributed locks, where each client can create a node with a unique identifier that represents its lock. When the client disconnects, the node is automatically deleted.

Watches

ZooKeeper supports the concept of watches. Client can set a watch on a znode. A watch will be triggered and removed when the znode changes. When a watch is triggered, the client receives a packet saying the znode has changed. If the connection between the client and one of the ZooKeeper server is broken, the client will receive a local notification.

API

One of the design goals of ZooKeeper is providing a very simple programming interface. As a result, it supports only these operations:

- *create* : creates a node at a location in the tree

- *delete* : deletes a node

- ***exists*** : tests if a node exists at a location
- ***get data*** : reads the data from a node
- ***set data*** : writes data to a node
- ***get children*** : retrieves a list of children of a node
- ***sync*** : waits for data to be propagated

Guarantees

ZooKeeper is very fast and very simple. Since its goal, though, is to be a basis for the construction of more complicated services, such as synchronization, it provides a set of guarantees. These are:

- **Sequential Consistency** — Updates from a client will be applied in the order that they were sent.
- **Atomicity** — Updates either succeed or fail. No partial results.
- **Single System Image** — A client will see the same view of the service regardless of the server that it connects to. i.e., a client will never see an older view of the system even if the client fails over to a different server with the same session.
- **Reliability** — Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- **Timeliness** — The clients view of the system is guaranteed to be up-to-date within a certain time bound.

Zookeeper | Software System Design | System Design Interview | Software Architecture | Software Architect

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium